COMMENT from Doug Lea's Malloc version 2.5

/*
A version of malloc/free/realloc written by Doug Lea and released to the public domain.

VERSION 2.5

* History:
  Based loosely on libg++-1.2X malloc. (It retains some of the overall
  structure of old version, but most details differ.)
  trial version Fri Aug 28 13:14:29 1992 Doug Lea (dl at g.oswego.edu)
  fixups Sat Aug 7 07:41:59 1993 Doug Lea (dl at g.oswego.edu)
    * removed potential for odd address access in prev_chunk
    * removed dependency on getpagesize.h
    * misc cosmetics and a bit more internal documentation
    * anticosmetics: mangled names in macros to evade debugger strangeness
    * tested on sparc, hp-700, dec-mips, rs6000
      with gcc & native cc (hp, dec only) allowing
      Detlefs & Zorn comparison study (to appear, SIGPLAN Notices.)

* Overview

This malloc, like any other, is a compromised design.

Chunks of memory are maintained using a `boundary tag' method as described in e.g., Knuth or Standish. The size of the chunk is stored both in the front of the chunk and at the end. This makes consolidating fragmented chunks into bigger chunks very fast. The size field also hold a bit representing whether a chunk is free or in use.

Malloced chunks have space overhead of 8 bytes: The preceding and trailing size fields. When a chunk is freed, 8 additional bytes are needed for free list pointers. Thus, the minimum allocatable size is 16 bytes. 8 byte alignment is currently hardwired into the design. This seems to suffice for all current machines and C compilers. Calling memalign will return a chunk that is both 8-byte aligned and meets the requested (power of two) alignment.

It is assumed that 32 bits suffice to represent chunk sizes. The maximum size chunk is 2^31 - 8 bytes. malloc(0) returns a pointer to something of the minimum allocatable size. Requests for negative sizes (when size_t is signed) or with the highest bit set (when unsigned) will also return a minimum-sized chunk.

Available chunks are kept in doubly linked lists. The lists are maintained in an array of bins using a power-of-two method, except that instead of 32 bins (one for each 1 << i), there are 128: each power of two is split in quarters. Chunk sizes up to 128 are treated specially; they are categorized on 8-byte boundaries. This both better distributes them and allows for special faster processing.

The use of very fine bin sizes closely approximates the use of one bin per actually used size, without necessitating the overhead of locating such bins. It is especially desirable in common applications where large numbers of identically-sized blocks are malloced/freed in some dynamic manner, and then later are all freed. The finer bin sizes make finding blocks fast, with little wasted overallocation. The consolidation methods ensure that once the collection of blocks is no longer useful, fragments are gathered into bigger chunks awaiting new roles.

The bins av[i] serve as heads of the lists. Bins contain a dummy header for the chunk lists. Each bin has two lists. The `dirty' list holds chunks that have been returned (freed) and not yet either re-malloc'ed or consolidated. (A third free-standing list contains returned chunks that have not yet been processed at all.) The `clean' list holds split-off fragments and consolidated space. All procedures maintain the invariant that no clean chunk physically borders another clean chunk. Thus, clean chunks never need to be scanned during consolidation.

* Algorithms

Malloc:

This is a very heavily disguised first-fit algorithm. Most of the heuristics are designed to maximize the likelihood that a usable chunk will most often be found very quickly, while still minimizing fragmentation and overhead.

The allocation strategy has several phases:

0. Convert the request size into a usable form. This currently means to add 8 bytes overhead plus possibly more to obtain 8-byte alignment. Call this size `nb'.

1. Check if the last returned (free()'d) or preallocated chunk is of the exact size nb. If so, use it. `Exact' means no more than MINSIZE (currently 16) bytes larger than nb. This cannot be reduced, since a chunk with size < MINSIZE cannot be created to hold the remainder.

This check need not fire very often to be effective. It reduces overhead for sequences of requests for the same preallocated size to a dead minimum.

2. Look for a dirty chunk of exact size in the bin associated with nb. `Dirty' chunks are those that have never been consolidated. Besides the fact that they, but not clean chunks require scanning for consolidation, these chunks are of sizes likely to be useful because they have been previously requested and then freed by the user program.

Dirty chunks of bad sizes (even if too big) are never used without consolidation. Among other things, this maintains the invariant that split chunks (see below) are ALWAYS clean.

2a If there are dirty chunks, but none of the right size, consolidate them all, as well as any returned chunks (i.e., the ones from step 3). This is all a heuristic for detecting and dealing with excess fragmentation and random traversals through memory that degrade performance especially when the user program is running out of physical memory.

3. Pull other requests off the returned chunk list, using one if it is of exact size, else distributing into the appropriate bins.

4. Try to use the last chunk remaindered during a previous malloc. (The ptr to this chunk is kept in var last_remainder, to make it easy to find and to avoid useless re-binning during repeated splits. The code surrounding it is fairly delicate. This chunk must be pulled out and placed in a bin prior to any consolidation, to avoid having other pointers point into the middle of it, or try to unlink it.) If it is usable, proceed to step 9.

5. Scan through clean chunks in the bin, choosing any of size >= nb. Split later (step 9) if necessary below. (Unlike in step 2, it is good to split here, because it creates a chunk of a known-to-be-useful size out of a fragment that happened to be close in size.)

6. Scan through the clean lists of all larger bins, selecting any chunk at all. (It will surely be big enough since it is in a bigger bin.) The scan goes upward from small bins to large. It would be faster downward, but could lead to excess fragmentation. If successful, proceed to step 9.

7. Consolidate chunks in other dirty bins until a large enough chunk is created. Break out to step 9 when one is found.

Bins are selected for consolidation in a circular fashion spanning across malloc calls. This very crudely approximates LRU scanning -- it is an effective enough approximation for these purposes.

8. Get space from the system using sbrk.

Memory is gathered from the system (via sbrk) in a way that allows chunks obtained across different sbrk calls to be consolidated, but does not require contiguous memory. Thus, it should be safe to intersperse mallocs with other sbrk calls.

9. If the selected chunk is too big, then:

9a If this is the second split request for nb bytes in a row, use this chunk to preallocate up to MAX_PREALLOCS additional chunks of size nb and place them on the returned chunk list. (Placing them here rather than in bins speeds up the most common case where the user program requests an uninterrupted series of identically sized chunks. If this is not true, the chunks will be binned in step 3 next time.)

9b Split off the remainder and place in last remainder. Because of all the above, the remainder is always a `clean' chunk.

10. Return the chunk.

Free:
Deallocation (free) consists only of placing the chunk on a list of returned chunks. free(0) has no effect. Because freed chunks may be overwritten with link fields, this malloc will often die when freed memory is overwritten by user programs. This can be very effective (albeit in an annoying way) in helping users track down dangling pointers.

Realloc:
Reallocation proceeds in the usual way. If a chunk can be extended, it is, else a malloc-copy-free sequence is taken.

Memalign, valloc:
memalign arequests more than enough space from malloc, finds a spot within that chunk that meets the alignment request, and then possibly frees the leading and trailing space. Overreliance on memalign is a sure way to fragment space.

* Other implementation notes

This malloc is NOT designed to work in multiprocessing applications. No semaphores or other concurrency control are provided to ensure fhat multiple malloc or free calls don't run at the same time, which could be disasterous. A single semaphore could be used across malloc, realloc, and free. It would be hard to obtain finer granularity.

The implementation is in straight, hand-tuned ANSI C. Among other consequences, it uses a lot of macros. These would be nicer as inlinable procedures, but using macros allows use with non-inlining compilers, and also makes it a bit easier to control when they should be expanded out by selectively embedding them in other macros and procedures. (According to profile information, it is almost, but not quite always best to expand.)

*/